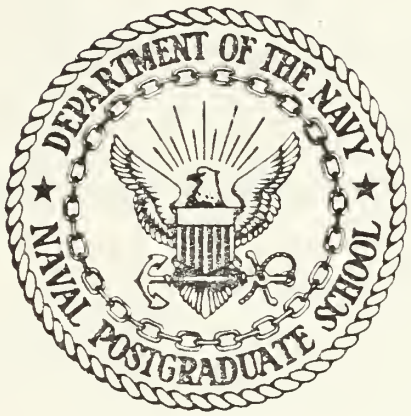
1984

An algorithm to test for confluence in a system of left to right rewrite rules.

Griffin, Rachel

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California



THESIS

AN ALGORITHM TO TEST FOR CONFLUENCE IN A SYSTEM OF LEFT TO RIGHT REWRITE RULES

by

Rachel Griffin
December 1984

Thesis Advisor: Daniel L. Davis

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* An Algorithm to Test for Confluence in a System of Left to Right Rewrite Rules		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis December 1984 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Rachel Griffin		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		12. REPORT DATE December 1984 13. NUMBER OF PAGES 62
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/ DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Rewrite rules, confluence

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Formal specifications of computation systems promise to afford us not only a strictly enforced discipline in describing the semantics of the system, but also a way of proving that the system functions as described. There exists a rich body of mathematical theory regarding the use of formal specifications. However, the bridge from the theoretical world to the practical world is nonexistent in most cases. This research (Continuned)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

concentrates on developing practical techniques to determine the correctness of a formal specification. The foundation for the specifications we describe is in abstract algebras. However, the view of the axioms for a specification is modified so that axioms are treated as left to right rewrite rules, rather than as mathematical equalities. The major result of the research is a new and practical algorithm to determine confluence in an axiom system of left to right rewrite rules that satisfy certain restrictions.

Approved for public release, distribution unlimited

An Algorithm to Test for Confluence in a System of Left to Right Rewrite Rules

by

Rachel Griffin
Lieutenant, United States Navy
B.A., Wellesley College, 1975

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1984

ABSTRACT

Formal specifications of computation systems promise to afford us not only a strictly enforced discipline in describing the semantics of the system, but also a way of proving that the system functions as described. There exists a rich body of mathematical theory regarding the use of formal specifications. However, the bridge from the theoretical world to the practical world is nonexistent in most cases. This research concentrates on developing practical techniques to determine the correctness of a formal specification. The foundation for the specifications we describe is in abstract algebras. However, the view of the axioms for a specification is modified so that axioms are treated as left to right rewrite rules, rather than as mathematical equalities. The major result of the research is a new and practical algorithm to determine confluence in an axiom system of left to right rewrite rules that satisfy certain restrictions.

TABLE OF CONTENTS

I. INTRODUCTION

A. BACKGROUND

There is a great deal of interest in the literature regarding the description of data types and programs in terms of formal mathematical models. Moreover, there is a growing interest in similarly specifying the machines we design so that we may describe the hierarchy of a computation system as a whole. These formal specifications, which take various forms depending on the researchers involved, share common goals:

- To describe only the essential qualities of a computation system without unnecessary detail.
- To provide a precise framework within which the power of mathematics may be used to support the specification model.
- To separate a description of a computation system from a specific environment. In other words, to provide for the portability of the system among diverse operational and theoretical environments.

Let us say that for some system, for example the commonly used data structure "stack", there exists a formal specification. With this specification we may consider the stack in its most abstract terms. The formal specification should describe all of the essential features of stack that we need to know in order to implement a stack on some machine, or describe its behavior in a particular program environment. However, the usefulness of a formal specification can be realized only if it is correct.

The main thrust of this research is to examine how we may determine, given a formal specification, whether or not it is useful. That is, can we through some practical means determine whether our specification describes a correct model of a computation system, or whether the specification itself will lead us astray? The most important ability that the use of a formal specification will give us is that we will be able to infer properties and truisms regarding the system apart from some physical representation or instantiation. We would like to say with some level of assuredness that a specific instance of a system is correct, and that it retains the essential qualities of the system we intended.

Of the many forms that formal specifications take in the literature, the view of formal specifications as abstract algebras appears to be the most promising. The work at the core of this research is found in Goguen (1978), Guttag (1978), and Bergstra and Tucker (1983). Any system to be described, be it a data type, program or physical resource, can be viewed as a set of values, and operations on those values. In a formal specification, one wishes to describe all of the legal values for a system and all of the ways that those values can be manipulated.

B. FORMAL ALGEBRAIC SPECIFICATIONS

An algebra is characterized by three components:

- A collection of sets called the carrier sets
- N-ary operations defined on the carrier set
- Distinguished elements of the carrier set, known as constants

The signature S of an algebra consists of the carrier set together with the operators. There may be equations E, which are axioms that equate valid terms of the algebra in some way. These equations address the behavior of the algebra. Axioms may or may not contain free variables to which arbitrary terms or expressions may be bound.

A formal algebraic specification is a template for algebras rather than a specific algebra itself. In the terminology of formal specifications, we speak not of the "carrier sets" but of "sorts". A sort is a name, or an index to a carrier set. We make this distinction as we do not wish to bias a particular algebraic specification to be the only possible description of a particular computation system. Expressions or terms of the specification are also templates for the individual elements of any carrier set satisfying the specification for the sort type of the expression. A single formal specification may represent a class of algebras satisfying the signature and axioms of the specification. By separating the "template" from the algebras, researchers in the area would like to say something about how specifications may be shown to be equivalent. It may be that two very dissimilar appearing specifications contain algebras in common which would demonstrate the equivalence of the formal specifications.

C. AN ALGEBRAIC SPECIFICATION FOR INTEGER

Figure 1.1 is an example of a formal algebraic specification for the INTEGER data type. What is important to note is the content of the sort(s), operations and the axioms rather than the syntax. Work concurrent to this research by Davis (1984), Yurchak (1984), and Lilly (1984) address the form of the formal specification.

```
Specification INTEGER
is
        Sort:  I
        Ops:
                0: -> I
                S: I -> I
                P: I -> I
                +: I, I -> I
        Axioms:
                P(S(x)) = x
                S(P(x)) = x
                +(P(x), y) = P(+(x,y))
                +(S(x), y) = S(+(x,y))
                +(0,x) = x
                +(x,0) = x
End INTEGER
```

The letter "I" denotes the sort, and represents any carrier set that satisfies the specification. "0", "S", "P", and "+" are the operators of the specification, and may be thought of as functions on the sorts. Thus "0" is a nullary operator, resulting in a value of sort "I", while "+" is a binary operator that accepts two arguments of sort I and returns a value of sort "I". "0" corresponds to the constant 0, while "S", "P", and "+" refer to the operations Predecessor, Successor and Plus. There is no limit on the number of sorts within a particular specification. Algebraic specifications may be used to describe a computation system of arbitrary size and complexity.

The formulation of terms in the specification may be thought of as a composition of the functions described by the operators. For example, "0" is a valid term in the specification. So then is "0 + 0" and "P(0 + 0)" and so forth. Let TERM(S) be the set of all valid terms formed by the operators on the sorts. Formally, TERM(S) may be constructed by a method known as the Herbrand

Construction. A full description is beyond the scope of this paper. The interested reader may consult Goguen (1978) for more detail. Suffice it to say that TERM(S) represents the "template" terms which will be mapped to the terms in a particular algebra satisfying the specification.

D. CONGRUENCE AND COMPUTABILITY

We are concerned with the congruences on the set of terms induced by the axioms. If there are no axioms for the specification, then the terms of the specification are distinct. They have not been related to each other in any way. However, the existence of axioms distinguishes equivalences between members of TERM(S). In our example specification for INTEGER, terms of the form +(P(x),y) are equal to the terms of the form P(+(x,y)). Berkstra and Tucker (1983) formally examined the properties that an algebraic specification must have in order to be useful, and describe the concepts of "initial" and "final" algebraic semantics. The formalism of their approach is beyond the scope of this paper, however the key idea behind their work is that for a given algebraic specification, we wish to determine whether or not the system is "computable". By computable, we mean that given two valid terms in the specification, we may effectively determine from the axioms whether or not the terms are equal. In terms of congruences, the axioms must partition the set TERM(S). We wish the nature of this partitioning to be decidable from a practical standpoint. If the system is not computable there will be questions regarding the congruence induced by the axioms involving specific terms that will go unanswered.

II. CONFLUENCE AND COMPUTABILITY

A. PRACTICAL COMPUTABILITY

Berkstra and Tucker (1983) emphasize "computability" as the key issue we need to examine regarding a specific formal specification. Plainly stated, a specification is computable if given any two legal terms in the algebra, there is an algorithm to decide that either the two terms are equal or they are not equal.

The theoretical work is of value in providing insight into the properties of these algebras with which we model our computation systems. Unfortunately, it does not give us a practical means to examine a specification and determine whether or not the object it describes is computable. If formal specifications are to be used in real situations, such theoretical qualities as computability must be determinable.

What do we mean in practical terms by a computable specification? We require of our specification that it model some real set of events. The specification describes a computation system that will have some physical representation according to a specific implementation. Those implementing the specification must be given a clear description of its behavior (i.e. the axioms) so that operations with the entity will proceed predictably and exactly in accordance with the specification. We cannot allow ambiguities regarding an implementation. Either two terms are equal, or they are distinct. Introducing a situation where the relationship between the terms is unclear may be tolerable in the theoretical world, but not something that can be left up to interpretation in a particular operational environment.

Let us examine an example which demonstrates the difficulty in determining whether or not a specification is computable.

B. AN EXAMPLE OF THE PRACTICAL COMPUTABILITY PROBLEM

Consider the following specification for a "nonsense" data type.

```
SPEC NONSENSE is
Sorts  N
Ops
        C: -> N
        /: N -> N
        +: N, N -> N
Axioms:
        1:      /(/(a)) = a
        2:      /(a) + C = a
```

Now consider a valid term in the specification, t = /(/(C)) + C. Mathematically, we do not impose any ordering on how axioms may be applied. In our hypothetical system, we may apply both axioms to the term. If axiom 1 is applied first, one resultant term is t' = C + C. If axiom 2 is applied first, then another result is t'' = /(C) + C. Axiom 2 may be applied to t'' a second time, and the final result is term t''' = C. Is C + C = C? Certainly. We know that these terms are the result of applying axioms to a common term. But let us say that we were given these two terms, C + C and C, and asked to decide whether or not they were equal, without the benefit of the above derivation. To answer the question, we start applying axioms to the terms. Our application of the axioms, which may proceed in either direction, would probably have to be done in a "hit or miss" fashion. For this simple example, we would undoubtedly find the common term quickly.

Will we be able to resolve the question of equality as quickly for any pair of terms? Consider our "hit or miss" process of applying the axioms to these arbitrary terms. Only if the two terms are equal, and only if we hit on the right combinations of axioms to apply will the process end. If the two terms are not equal to each other, we will never be able to say so with any conviction. Our simple specification is not practically computable. For example, the reader may try to answer the question:

$$\text{Is } /(C) = /(\ /(C) + (C + C))?$$

The author does not know, and prefers to find a technique other than trial and error to answer such a question.

C. CONFLUENCE OF LEFT TO RIGHT REWRITE RULES

Alan Bundy (1983) discusses the concept of confluence for a set of left to right rewrite rules. Bundy's work is principally in the areas of mathematical reasoning and theorem proving, yet provides a practical way to look at algebraic specifications. He defines confluence as follows:

> "A set of rules is confluent if whenever an expression, exp, can be rewritten in two different ways, say to int1 and to int2, then int1 and int2 can both be rewritten to some common rewriting, comm."
>
> Bundy (1983)

Bundy's meaning in discussing the "rewriting" of a term is that there are left to right rewrite rules which may be applied to an arbitrary term in the system. For example, there may exist a rule

$$X.0 \rightarrow 0$$

where X is a free variable for some system. We may rewrite the term 2.0 to 0 by this rule.

Confluence is exactly the property we wish to have for our specification. Applying axioms to a given term is very much like applying the rewrite rules in Bundy's system. Transforming a term to some other term in accordance with the axioms is actually a step by step process, determined by applying the axioms in some sequence, one at a time. Bundy's rewrite rules work left to right exclusively. A term that is the result of a sequence of left to right transformations is equal to the starting term in the same way that we might think of it had the sequence been performed right to left from the resultant term. Applying the axioms only left to right provides us with a sort of anchor with which we may examine the properties of the axioms.

Let us examine what happens to a given term under an algebraic specification where we consider our axioms to be left to right (L->R) rewrite rules, rather than mathematical equalities that may be applied in either direction. Given a term t, either there exists a rewrite rule that applies, or there does not. If there is, the term may be rewritten in one step to one or more terms,

depending on how many of the L->R rules applied to t. Let us assume for the moment that these rules are terminating. This means that there are no terms to which axioms may be applied indefinitely. Therefore, for a given term, there will come a point when the term is transformed to some t', to which no rewrite rule applies. This set of terms to which no axioms apply forms a sort of kernel at the heart of all the terms in the specification. All terms in the specification to which one or more of the axioms apply will be transformed to one (or more) of the terms to which no axioms apply. In order to satisfy the definition of confluence, all terms which are equal under the axioms of the specification will be transformed to one and only one term in that set of terms to which no axioms apply.

Returning to our earlier example of the NONSENSE specification, let us think of the axioms in the specification as L->R rewrite rules. The original term, /(/(C)) + C , was transformed to two distinct terms, t' = C + C and t'' = /(C) + C. Neither t' nor t'' can be rewritten to any other terms. Therefore, the conditions for confluence fail. We see that our rules as they currently exist, are not confluent.

We will treat all axioms of a given specification as L->R rewrite rules. It may be argued that one loses mathematical generality by imposing such a restriction, however, the L->R rewrite rules are more like the computational techniques that may be found in a real machine. When a computer adds two numbers together, or performs some sort of string manipulation, there exists at its operational heart, either in hardware or software, a set of rules which are followed to perform the computation. For example, if we add 3 + 6, we fully expect to see a result of 9. It would be surprising to see an answer of "11 - 2", although mathematically "11 - 2" is every bit as valid as is the answer "9". The point is, we expect our computations to result in some normalized version. We may think of this normalized version to be the simplest, or the shortest, or the most conventional, depending on our experience. We do not require of our computers that they (outside of the AI environment) embark on a computational search without end. In short, we wish our answers to be in a normal form.

D. CANONICAL FORMS

Consider a set of terms and a set of axioms on those terms. We discussed earlier that the terms will fall into two categories. For each term, either one or more of the L->R rewrite rules apply, or no rewrite rules apply. The set of terms for which none of the rules apply is indeed a normal form. But what of the case where the rules are not confluent. Which normal form do we choose as our result? Do we output all of them and ask the user to decide? Certainly not. At the heart of computability is the requirement that we may be assured that equality of terms is decidable, without ambiguity.

Our goal is that the set of normal forms for a particular set of terms be canonical. Bundy set forth the following definition:

> "A set of rules is canonical when all of the normal forms of each expression are identical. That is, it does not matter how you go about applying the rules to an expression, you will get the same result. The common normal form is called the canonical form of the starting expression."
>
> Bundy (1983)

Bundy also showed that a confluent set of rules is canonical. Can we determine whether or not the axioms of a formal specification are confluent? If an effective method could be found to determine the confluence of a set of L->R rewrite rules, there would then exist the means to determine the practical computability of a formal specification. Confluence implies computability, because the issue of whether term t is equal to term t' is decidable.

III. AXIOM SYSTEMS AS LEFT TO RIGHT REWRITE RULES

A. THE METHOD

An axiom testing program provides empirical support for this research. Initially the program was built from a desire to model the computation of terms in a given algebra, that we might examine the properties that a true specification testing system would need to have. From this beginning, grew a system with which we are able to examine properties regarding the reduction of terms in confluent and non-confluent systems of axioms and to develop a practical method of testing for the confluence of axioms.

In the following pages, the research is summarized and important terminology introduced. Aside from providing the reader with the facts demonstrated by the research, an important goal of this chapter is to describe the reasoning which leads to the establishment of those facts.

B. TERM MEMBERSHIP

The signature of a formal specification, consisting of the sorts and the operators on the sorts, specifies all of the valid terms for that specification. The question of term membership is decidable. It can be shown that for every specification, the generation/recognition of terms can be accomplished with an LL(1) grammar (Davis(1984)). Once the sorts and operators on the sorts have been established, there exists a mechanical way to recognize the elements of the specification.

C. TERMS AS EXPRESSION TREES

The next task is to examine the possibility of determining whether or not two arbitrary terms are provably equal or not equal according to the specification. If a specification consists of only terms with no axioms, then all terms are distinct. No term can be proved equal to another, because there are no axioms to transform one term to another. The entire set of terms forms a canonical set, as the conditions for a confluent set are trivially satisfied.

The terms themselves are naturally viewed as expression trees. Each term is expressed as a function or the composition of functions corresponding to the operators of the specification. Let us again examine a simple specification for the integers.

```
SPECIFICATION  INTEGER
is
        SORT:       INT

        OPS:
                0: -> INT
                S: INT -> INT
                P: INT -> INT
                +: INT, INT -> INT
        AXIOMS:
                +(x, 0) = x
                +(0, x) = x
                +(P(x), y) = P(+(x,y))
                +(S(x), y) = S(+(x,y))
                P(S(x)) = x
                S(P(x)) = x
```

The axioms of the INTEGER specification are well supported by mathematical theory and provide us with a secure starting point for the examiniation of axioms that we knew to be correct and representative of the entity they describe. Recall that all axioms will be treated as L->R rewrite rules, rather than strict mathematical equalities. Thus the "=" between the two terms of any given axiom should be viewed as a "->". "0" corresponds to the constant 0, "P" to the predecessor function, "S" to the successor function, and "+" corresponds to addition.

D. THE AXIOM TEMPLATE

The axiom testing program accepts the terms of the INTEGER specification in postfix form, and builds the tree corresponding to the term. The term is transformed using a recursive algorithm which examines the root of the tree and determines whether or not one of the axioms applies. If some axiom does apply, the transformation is made, and the process begun again. If no axiom applies, the tree is traversed in postfix, and each node of the tree examined in turn for the possibilty of its being the root of an axiom tree. We now introduce the notion of

an "axiom template". Consider the axiom P(S(x)) -> x. The template of this axiom is the tree formed by the three elements of the left hand side of the axiom. The root of the template tree is at P, and the leaf is the free variable, x. In order to apply that or any other axiom, the template is in effect "overlayed" on top of some part of the expression tree of the term you are examining. If there is a match of the operators of the expression tree and the template tree, then a particular axiom applies. Any free variables are bound to whatever value follows its parent template operator in the expression tree. This matching of axiom template to the nodes of the expression tree represents a mechanical way to compute the terms of the INTEGER specification. Only when there is a match, operator for operator, between a single axiom and a particular subexpression of the term in question, can a particular axiom be applied. Figure 3.1 illustrates the axiom templates for the INTEGER specification.

E. APPLYING AXIOMS TO EXPRESSION TREES

Looking at the axioms for INTEGER, it is obvious that the terms "0" and P(S(0)) are equal under the axioms. "0" is irreducible, while the expression tree formed by P(S(0)) contains a subtree over which the template for the axiom P(S(x)) may be overlayed. In this case, the free variable x will be bound to 0. Similarly P(S(P(S(0)))) contains two subtrees over which the axiom template P(S(x)) may be overlayed, and therefore two places where that axiom may be applied. But notice that while that axiom applies twice to P(S(P(S(0)))), the template for S(P(x)) may also be matched to the tree, beginning at the first S node in the expression tree. The templates overlap in this particular expression. This overlapping of templates may be observed in an infinite number of arbitrary INTEGER terms. The phenomenon of template overlap becomes very important in examining the reduction of an arbitrary term by the axioms.

F. INITIAL RESTRICTIONS ON THE REWRITE RULES

In order to investigate the behavior of a set of axioms on a set of terms, we place certain restrictions on the axiom system. First of all, the axioms are viewed as L->R rewrite rules as previously described. Certainly some mathematical

Specification INTEGER

Axiom Template	Axiom
P | S | x	P(S(x)) = x
S | P | x	S(P(x)) = x
+ / \ P y | x	+(P(x), y) = P(+(x,y))
+ / \ S y | x	+(S(x), y) = S(+(x,y))
+ / \ 0 x	+(0,x) = x
+ / \ x 0	+(x,0) = x

Axiom Templates for the INTEGER Specification
Figure 3.1

generality is lost, but the computation of terms by a left to right rewrite system is nonetheless illustrative of the reduction of one term to another.

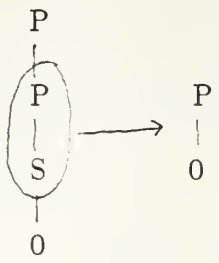
An additional restriction is that variables will be preserved. This means that if a free variable is on the left hand side of an axiom. that same free variable will appear on the right, though of course, its position in the tree is in accordance with the transformation specified by the axiom. Moreover, there will appear no free variable on the right hand side of an axiom that did not appear on the left. The implication of this restriction is that for any given term, the application of a particular axiom will not affect the subtree bound to any free variables of the axiom template. Note also that we deny conditional axioms such as

$$x + y = y + x \qquad \{ x \mathrel{!}= / \}$$

Therefore, whatever transformation the axiom dictates will not affect any of the subtrees of the term that are not specifically part of the template.

G. LOCALIZATION OF AXIOM TRANSFORMATIONS

Note that there are now two important properties regarding axiom effect that we may rely on in predicting how a particular axiom will transform an arbitrary term. First, we know that subtrees bound to free variables of an axiom template remain unaffected by the transformation. Second, since the template matches a particular subtree of the expression tree, and transforms that expression tree into an equivalent form, all ancestors (if any) of that subtree remain unaffected. For example, the expression P(P(S(0))) contains a subtree P(S(0)) which matches the template of the axiom P(S(x)) -> x. The root of the original expression is not part of the template, and the resultant expression will be rooted at the instance of the operator P. The effect of a particular axiom can therefore be thought of as "localized" to the nodes of the expression tree that specifically match a given template. Ancestor nodes of the expression tree will be unaffected, as well as all descendants bound to the free variables of the axiom template. In our previous example, the first "P" remains in the expression tree, as will the instance of "0". The axiom transformed the tree by dropping the subtree P(S(..)), leaving the argument to "S". The resultant term in this case is P(0). See Figure 3.2. In this example, no further axioms apply. For an arbitrary term in the set of valid terms for the specification, there may be other subtrees to which axioms apply.



Localization of Axiom Effect
Figure 3.2

H. DEFINITIONS

The following terms and notation will be used throughout the discussion:

- S -- The signature of the specification, consisting of the sorts along with the operators of arity n, n >= 0.

- E -- The set of axioms of the specification. The number of axioms is some finite number n. We regard all axioms as L->R rewrite rules.

$$E = \{ e(i) \mid e(i) = t \text{ -> } t', t, t' \text{ in } TERM(V), 1 <= i <= n \}$$

- **Specification** -- A specification is a pair (S, E) where S refers to the sorts and operators of the specification, and E is the set of axioms on those operators.

- **TERM(S)** -- The set of all valid terms in the specification. We know that membership in TERM(S) is decidable for any arbitrary term.

- **TERM(V)** -- The set of all valid terms with free variables. Note that membership is also decidable. Note also that TERM(S) is a subset of TERM(V). The difference between any term in TERM(S) and term in TERM(V), is there exist terms in TERM(V) which form trees which have variables for leaves. No internal nodes may be variables. The leaves of TERM(S) are all nullary operators.

- **AXIOM TEMPLATE** -- The expression tree formed by the left hand side of some axiom. In order to apply the axiom to an arbitrary term in TERM(V), the template must be matched exactly to some subtree of the term, matching operator for operator, and binding free variables to subtrees

of the template found in the arbitrary expression.

- **TERM(I)** -- The set of all irreducible terms in the specification. In other words, for all t in TERM(I), there exists no subtree in t that matches any of the axiom templates in E. TERM(I) is a subset of TERM(S).

- **TERM(R)** -- The set of all terms in the specification for which one or more axioms apply.

- **AXIOM OPPORTUNITY** -- For some term t, an axiom opportunity is a subtree of t, that matches the template for some axiom in E.

- **APPLICABLE(t)** -- For some t in TERM(V), the set of all axiom opportunities that exist in the expression tree. APPLICABLE(t) = { a(i) | a(i) is an axiom opportunity; i >= 0 }

- **t ->a(i) t´** -- t in TERM(V) is transformed to t´ in TERM(V) by the application of a L->R rewrite rule represented by a(i) in APPLICABLE(t).

- **t ->+ t´** -- t in TERM(V) is transformed to t´ in TERM(V) by some sequence of one or more transformations by L->R rewrite rules in E

- **t ->* t´** -- t in TERM(V) is transformed to t´ in TERM(V) by some sequence of 0 or more transformations by L->R rewrite rules in E

- **t |-- t´** -- t in TERM(S) is provably equal to t´ under the axioms of the specification. t |-- t´ if and only if t ->* t´ or t´ ->* t

Can we develop an algorithm for confluence? We know that if the the axioms can be shown to be confluent, then the terms in TERM(I) consitute a canonical set. Certainly, the first method that comes to mind is to actually test each member of TERM(S), over the given set of axioms. If there exist no terms that can be transformed to two or more distinct terms in TERM(I), then the axioms are confluent. Unfortunately, for all but the most trivial specifications, this is impossible. One cannot test an infinite number of terms. We must concentrate on examining the behavior of the axioms as they are applied to an arbitrary term, in the hope of identifying qualities that may indicate a finite method of testing. The rich body of theory is of little value, and less abstract means of evaluating an axiom system must be found.

I. REWRITE RULES MUST TERMINATE

In addition to the aforementioned restrictions, we wish our axioms to have another property. Consider the expression tree for some term. If there is a match to one of the axiom templates, the axiom may be applied and the tree transformed. We must have an assurance that no term can be transformed indefinitely. In other words, we cannot have a term such that no matter how many transformations are made, there is always another axiom that may be applied. Such a situation would exist in the following example.

$$x + y = /(x + y)$$

Mathematically, there is nothing wrong with such an axiom. It merely establishes that terms of the form of the left hand side are equal to terms of the form of the right hand side. But notice that when the axiom is regarded as a L->R rewrite rule, given a term that satisfies the left hand condition, the transformation of the term by the axiom will result in another term which also provides a match for the axiom template.

In the case of the above example, the right hand side of the axiom formed its own template. Although the axiom may be correct mathematically, we cannot allow this condition in our axiom system. This restriction will limit the power of the axioms as there will be useful properties of a system which cannot be specifically represented in a system of L->R rewrite rules. The most obvious example of this is the property of commutativity:

$$x + y = y + x$$

The only node in the template expression is "+", while x and y are dummy variables. Given that this axiom may be applied once in a particular expression tree, it may be applied an infinite number of times. The INTEGER specification shown above is not, however, crippled by this restriction. Given any term which is valid under that specification, it should be obvious that every term in the integers can be expressed. The above axioms will allow an arbitrary expression to be reduced to its canonical form. In the case of the integers, this canonical form is 0, some sequence of P(...(0)...) or some sequence of S(...(0)...), corresponding to 0, the negative and the positive integers.

Just as no axiom may include its own template in the right hand term, given a set of axioms, there must exist no cycles of templates and right hand side terms. This condition can be illustrated by the following specification:

```
SPEC NONSENSE
is
        SORT N
        OPS
                C: -> N
                /: N -> N
                +: N, N -> N
                *: N,N -> N
        AXIOMS
                1:      /(x + y) = x * y
                2:      x * y = /(x) + y
                3:      /(x) + y = /(x + y)
```

Notice that by themselves, none of the axioms will start an infinite sequence of reductions. However, the transformation of some expression by axiom 1 will result in a term which immediately qualifies for transformation to another term under axiom 2. Axiom 2 in turn, causes the transformation of the tree to a form which will qualify for its reduction to a term under Axiom 3, which will ultimately result in a term which is now qualified to be transformed under Axiom 1.

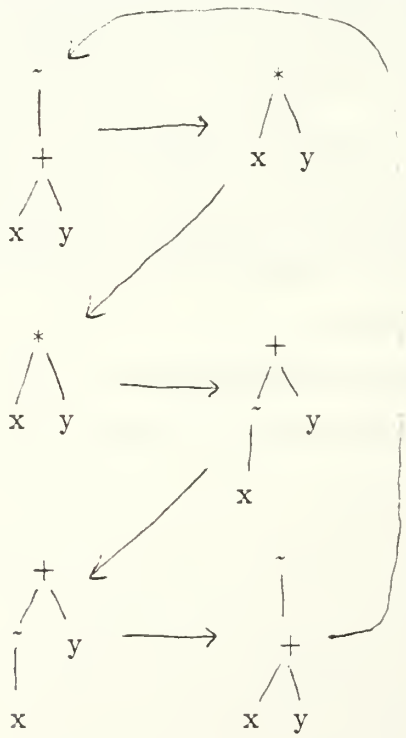
J. TEST FOR THE TERMINATION PROPERTY

Thus we must add the restriction that the axioms be terminating. There is a simple test for termination. We know that the existence of a subtree that matches one of the axiom templates will allow a transformation under that axiom. First examine the axioms to see if any right hand side matches its own template. Then examine the list of axioms for a "template loop". Visually, one may consider the left hand and right hand expressions of an axiom as nodes in a digraph. For each axiom, draw an arc from its left hand side to its right hand side. Examine each right hand side for the possibility that it provides a match for one of the other left hand side templates. If it does, draw an arc from it to the corresponding left hand side. Continue in this manner for all of the left hand side nodes. If at the end of the procedure, there exist any cycles, then the set of

axioms is non-terminating. Figure 3.3 illustrates how our specification NONSENSE violates the termination condition.

K. COMPUTATION PATHWAYS

We know that given an arbitrary term t in TERM(S), either the term is in TERM(I) and there are no axioms which may be applied to the term, or the term is in TERM(R), and one or more axioms may be applied. Consider the terms that may be transformed. If one axiom applies, then there is only one path of length one for the term transformation to take. Apply that one axiom, and examine the resultant term for its membership in TERM(R) or TERM(I). If n axioms apply, which is to say that there are n elements in APPLICABLE(t), then there are at least n different sequences of transformations of the term. For



Test for the Termination Condition
There is a cycle, thus the axioms are not terminating.
Figure 3.3

example, consider an expression tree in which there are three subtrees that match some axiom template. There are at least three sequences of transformations of the term, depending on which of the subtrees you choose to transform first. We say "at least" three sequences, because at this point, we are saying nothing about where in the tree the three axioms apply, nor are we concerned with what axioms may apply as a result of transforming the term on one of the three initial axiom opportunities. We now introduce the notion of computation pathways and partial computation pathways.

COMPUTATION PATHWAY:

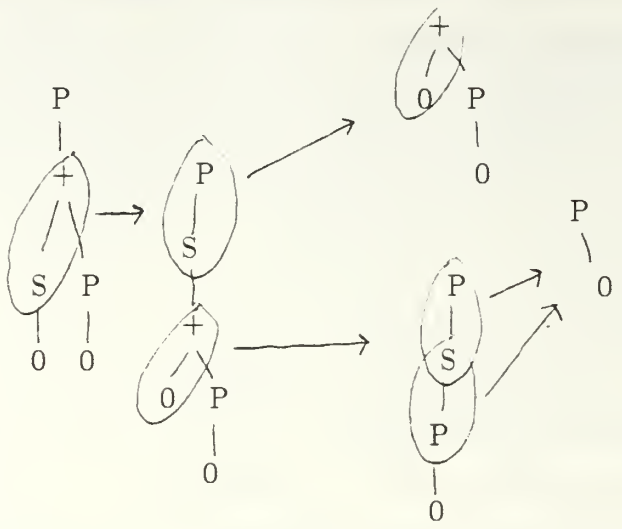
t ->* t´ t in TERM(S) and t´ in TERM(I)

PARTIAL COMPUTATION PATHWAY:

t ->* t´ t in TERM(S) and t´ in TERM(R)

For a given term t, a computation pathway refers to some sequence of transformations by the axioms such that t is transformed to an expression to which no axioms apply. A partial computation pathway for t is some sequence of transformations to some term t´, for which additional axioms may apply. It should be noted that each term in the computation or partial computation pathway is provably equal to each of the other terms.

The order of axioms applied is important. Consider a term for an arbitrary specification to which both axioms 1 and 2 of the specification apply. Let APPLICABLE(t) = { a(1),a(2) }, where a(1) is the opportunity to apply Axiom 1 and a(2) is the opportunity to apply Axiom 2. There exist at least two computation pathways, one beginning with Axiom 1 and another beginning with Axiom 2. Upon choosing either Axiom 1 or Axiom 2, we may or may not be able to apply the other on the resultant term. In other words, consider the transformation t ->a(1) t'. The original opportunity to apply Axiom 2, a(2), may or may not exist for a transformation t' ->a(2) t''. Whether or not the application of one axiom denies the opportunity to apply another is significant.



Exhaustive Computation of term P(S(0) + P(0))
There are 3 complete computation pathways
Figure 3.4

L. EXHAUSTIVE COMPUTATION OF A TERM

We must identify a method of testing and a set of items to be tested. The axiom testing program is able to exhaustively compute a given term. Given a specific term, the program stores the information regarding the axioms currently applicable, APPLICABLE(t). By means of a recursive procedure, the term is transformed according to each of the axiom opportunities, and the resultant term examined in turn for its APPLICABLE(t). Figure 3.4 illustrates the exhaustive computation of a term from the INTEGER specification.

Even for simple terms of a specification, the number of computation pathways can be enormous. Exhaustive computations of arbitrarily long terms could take a very long time. Not only should our test set have a manageable number of elements to be tested, but each of the elements should have as small a set of computation pathways as possible. Figure 3.5 illustrates several terms from the INTEGER specification, their resultant terms under the axioms, and the number of computation pathways for each term.

Specification is INTEGER

Operators

0: -> A
P: A -> A
S: A -> A
+: A,A -> A

The Axioms for the INTEGER Specification:

1: P(S(x)) = x
2: S(P(x)) = x
3: +(P(x), y) = P(+(x,y))
4: +(S(x), y) = S(+(x,y))
5: +(0, x) = x
6: +(x, 0) = x

Expression: P(S(0 + 0)
Computed: 0
Number Computation Pathways = 4

Expression: (P(0) + S(0)) + (P(0) + 0)
Computed: P(0)
Number Computation Pathways = 407

Expression: (0 + 0) + (P(0) + 0)
Computed: P(0)
Number Computation Pathways = 76

Expression: (0 + 0) + P(S(P(0) + S(0)))
Computed: 0
Number Computation Pathways = 252

Expression: P(S((0 + 0) + (0 + 0)))
Computed: 0
Number Computation Pathways = 96

Exhaustive Computations of Selected INTEGER Terms
Figure 3.5

M. THE INTERACTION OF AXIOM TEMPLATES

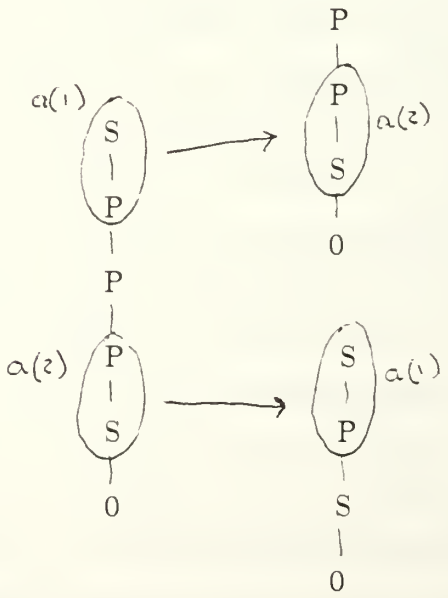
Examining the computation pathways offers a sound method of approaching a practical test for confluence. Unfortunately, the determination of which terms out of an infinite number of terms to choose, is the stumbling block. We observed earlier that the effect of an axiom transformation is localized to the template operators of the expression. Neither ancestors nor descendents of the template operators will be affected. Consider the following term from the INTEGER specification:

$$S(P(P(P(S(0)))))$$

Currently, two axioms apply to the expression, $S(P(x)) = x$ and $P(S(x)) = x$. Observe that because the opportunities to apply the axioms are separated in the expression, no matter which axiom we choose to begin the transformation, the other axiom will still apply to the transformed term:

Let a(1) be the opportunity to apply $S(P(x)) = x$
Let a(2) be the opportunity to apply $P(S(x)) = x$

1: $S(P(P(P(S(0)))))$ -> a(1) $P(P(S(0)))$
2: $S(P(P(P(S(0)))))$ -> a(2) $S(P(P(0)))$

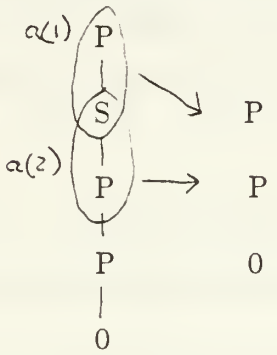


In this example, a(2) is an element of APPLICABLE(P(P(S(0)))), while a(1) is an element of APPLICABLE(S(P(P(0)))). Applying one axiom over another did

not deny the opportunity to apply the one not chosen in a later transformation. However, observe the case where the templates overlap as in the following term:

P(S(P(P(0))))
Let a(1) be the opportunity to apply P(S(x)) = x
Let a(2) be the opportunity to apply S(P(x)) = x

1: P(S(P(P(0)))) -> a(1) P(P(0))
2: P(S(P(P(0)))) -> a(2) P(P(0))



Note that applying a(1) or a(2) results in a term for which the axiom opportunity not initially chosen no longer applies. In this case, all is well, because both computation pathways result in the identical term in TERM(I). However, this is not always the case, as illustrated in Figure 3.6.

N. THE DEVELOPMENT OF A TEST SUITE

Let us further examine how the axioms interact with each other. We know that the templates dictate the only possible situations where axioms will apply to an expression. We also have observed that if the axioms do not interact with each other in a given expression, (i.e. there is no sharing of template operators) the effect of the application of a particular axiom is localized to a spot in a tree which does not affect the application of other axioms.

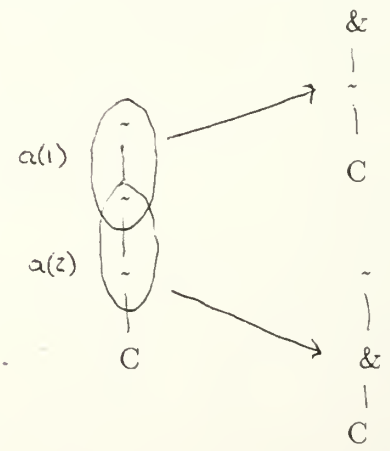
We would like to show that the axiom templates provide all of the information necessary to identify a set of test cases. If there is no subtree of an expression tree that matches any of the axiom templates, then there is no possible way to apply an axiom. If the subtrees where there are opportunities to apply axioms are separated, and do not share any template nodes, then applying one axiom over another will not deny the application of any of the initially

Specification is NONSENSE
Operators
C: -> A
˜ : A -> A
&: A -> A
+: A,A -> A
The axioms for the NONSENSE Specification are:

1: ˜ (˜ (a)) = &(a)
2: C + a = a
3: ˜ (a) + b = ˜ (a + b)
4: a + ˜ (b) = ˜ (a + b)

Consider expression t = ˜ (˜ (˜ (C)))

APPLICABLE(t) consists of two opportunities to apply Axiom 1.
APPLICABLE(t) = { a(1), a(2) } where
a(1) is the opportunity to apply Axiom 1 at the root of the expression and
a(2) is the opportunity to apply Axiom 1 at the second node in the expression tree
1: ˜ (˜ (˜ (C))) -> a(1) &(˜ (C))
2: ˜ (˜ (˜ (C))) -> a(2) ˜ (&(C))
The two resultant expressions &(˜ (C)) and ˜ (&(C)) are not provably equal under the axioms.



Divergent Computation Pathways
Figure 3.6

applying axioms. The problem is the case where the templates overlap with each other, as we have seen in previous examples.

For each axiom template, consider all of the ways that it may share template nodes with one or more of the other templates, (including itself) such that its root remains the root of the expression tree. This is a finite number, as there are a finite number of axioms, and each has a finite number of operators which may be shared. For example, the INTEGER axiom P(S(x)) may share template operators with other templates in only one way, P(S(P(x))), where it shares an operator with Axiom S(P(x)). Axiom +(P(x), y) may share template operators in only 3 ways:

P(S(x) + y	Shares operators with Axiom 2
P(x) + 0	Shares operators with Axiom 6
P(S(x)) + 0	Shares operators with Axioms 2 and 6

Note that we may only share operators of the current axiom being examined. It is true that another overlap situation exists with the expression

P(S(P(S(x)))) + y Overlap of Axioms 1 (twice),2,3

However, in that case, it is not our original axiom that shares operators with all of the overlapping axioms, but subtrees of the expressions that have been formed by an axiom that overlaps the original.

While it may seem that such a process of examination will result in only a very few overlapping situations, when there are an infinite number of ways that the set of axioms may overlap, we will see that these are the only cases you need examine. Consider that for some axiom template, there are a finite number of ways that it may overlap with one or more of the axioms. If we determine that none of these overlap situations results in more than one irreducible term, and we do so for all of the axioms, then the axioms are confluent.

In terms of the tree transformation, non-confluence is observed when at some point in the exhaustive computation of a term, the path "forks" to two (or more) different resultant terms. Where can such forking occur? Only at a place where the decision to use one axiom denied the opportunity to apply another or a set of others. Recall that the application of an axiom will not affect the nodes of the

expression tree that are ancestors of the root of the template. For every term reduced by axioms which are not confluent, the forking to pathways ending in different resultant terms begins at some subexpression of the tree whose root is the root of an axiom template. So we need not be concerned with arbitrary ancestors to the overlapping templates.

Why is it that examining only the possible overlap situations for each of the axiom templates will be sufficient? There are certainly cases where the axioms overlap one over another forming "chains", many axioms long, down a tree. For example, the term

$$P(S(P(S(P(S(P(S(0))))))))$$

from the INTEGER specification demonstrates that beginning with the root, there are overlapping axiom opportunities all the way down through the tree until the "0" node? If we go by the above method, the only axiom overlap cases that would be checked out for $P(S(x)) = x$ and $S(P(x)) = x$ with respect to each other, would be

$$P(S(P(x)))$$
$$S(P(S(x)))$$

If these overlap situations have been examined and no divergent pathways found, then consider the above expression. No matter where in the tree you choose to start applying your axioms, the root of the subexpression affected by that axiom will be either P or S. The only axiom opportunities that can be denied by the application of the axiom you choose are those that overlap the axiom opportunity template of the axiom you decide upon. We know that free variables are preserved, so that the expression bound to the variables of either overlap template situations or original axioms remain unaffected. If you choose to apply an axiom and there does occur a branching to two or more different resultant terms, then the specific place along the computation pathway can be identified to a spot where there was overlap between the axiom chosen and the other opportunities which overlapped with that axiom opportunity. If all of the template overlap situations are examined and found to cause no divergent pathways, then there can be no place in an arbitrary expression where the

pathways diverge. No matter where in the tree you apply your axiom, and no matter how long the "chain" of overlapping axiom templates the computation pathway at any particular point depends only upon the axiom you choose to apply, and the axioms it may or may not have denied.

We now have the basis to develop a "test suite" for a given set of axioms. We simply need to identify all of the ways that each of the axioms can overlap with each of the others. That is a simple task which was easily programmed into the axiom testing program, and whose algorithm will be given in Chapter IV. Let us define TESTSUITE(E) to be the set of all axiom overlap terms formed by that algorithm, which are simply terms in TERM(V) that represent all of the ways that axioms may overlap. We know there exists an effective procedure to exhaustively compute a term, which is to examine every possible computation pathway for a term in TERM(S). But our overlap templates contain free variables, which of course can represent an infinite number of subtree expressions. Are we back to the problem of literally testing every term?

No. We know that variables are preserved. We wish to test the interaction of the specific axiom templates which overlap in a given element of TESTSUITE(E). Choose some t in TERM(S) which is irreducible, that is, some term for which no axiom applies. Substitute that term for each of the free variables in TESTSUITE(E). Now each of the terms is in TERM(S) and may be exhaustively computed. If the specification is multi-sorted, then for each sort represented by a free variable in TESTSUITE(E) choose some t in TERM(I), and make the appropriate substitutions. If for every term, every computation pathway results in the same resultant term in TERM(I), you have shown that no matter how the axioms interact, there will be no forks to different irreducible terms, and thus the axioms must be confluent.

IV. A PRACTICAL TEST FOR THE CONFLUENCE OF AXIOMS

We are ready to formally present the algorithm. First we outline the major results of the research that support the algorithm. Next the algorithm is presented. Last, we prove the validity of the algorithm.

A. THEOREM 4.1

1. Statement

Let (S,E) be a specification.

E is a set of unconditional, terminating L->R rewrite rules such that variables are preserved.

Let t be in TERM(S) and a(i), a(j) be in APPLICABLE(t).

If there exists no sharing of operators between a(i) and a(j), then

t -> a(i) t' ==> a(j) is in APPLICABLE(t') and
t -> a(j) t'' ==> a(i) is in APPLICABLE(t'')

2. Proof

We know the effect of an axiom transformation is localized to the operators of the axiom templates. Since a(i) and a(j) represent two instances of axiom templates which do not overlap, no matter what happens to the expression tree during the transformation on either a(i) or a(j), there can be no change made to any of the operators which contribute to the other opportunity. Simply stated, in this situation the application of the axiom represented by one of the axiom opportunities cannot deny the future application of the other.

B. THEOREM 4.2

1. Statement

Let (S,E) be a specification.

E is a set of unconditional, terminating L->R rewrite rules such that variables are preserved.

Let t, t' be in TERM(S), and TESTSUITE(E) be empty.

t -> a(i) t' ==>
APPLICABLE(t) \ {a(i)} is a subset of APPLICABLE(t')

2. Proof

We know in this case the axioms do not overlap. Thus there can be no term in TERM(S), such that there exists two or more axiom application opportunities that share template elements.

We can now say, using Theorem 4.1, that for each pairing of axiom application opportunities, transforming a term on one of the opportunities will not deny the opportunity to apply the other. This is true for all the pairs.

For some a(i) in APPLICABLE(t), and |APPLICABLE(t)| = n,

t -> a(i) t' ==> a(1) in APPLICABLE(t') and
...
a(i - 1) in APPLICABLE(t') and
a(i + 1) in APPLICABLE(t')
...
a(n) in APPLICABLE(t')

Thus, APPLICABLE(t) \ {a(i)} is a subset of APPLICABLE(t)

C. THEOREM 4.3

1. Statement

Let (S,E) be a specification.

E is a set of unconditional, terminating L->R rewrite rules such that variables are preserved.

Let t be in TERM(R), t' be in TERM(S), and | APPLICABLE(t) | = n

If TESTSUITE(E) is empty then there exists n! computation pathways of length n, representing every possible sequence of transforming t by the n elements of APPLICABLE(t), and for each computation pathway

t ->+ t'

2. Proof

APPLICABLE(t) = {a(1)...a(n)}. We know that there are n! permutations of n items. Axioms may be applied in any order, thus we may choose whichever axiom opportunity we wish to apply to a given expression tree. Number the axiom opportunities from 1 to n. Enumerate the n!

permutations of the axiom opportunities. Construct the computation pathway for each permutation, for example:

t -> a(1) t'
t'-> a(2) t''
...
t''' -> a(n) t''''

How can we establish that for all the computation pathways, the resultant t' is the same term? The effect of each transformation is localized to some subtree of the original term, and we know there will be no overlapping of templates. Since the effect of a particular axiom application opportunity must be the same whenever it is applied, no matter what order we choose to apply the opportunities, the overall effect is the same. All resultant terms must be equal to each other, because if they are not, this implies that some a(i) produced two different local effects. We know this cannot happen, as for each a(i), the local effect is precisely determined by the axioms.

D. THE ALGORITHM FOR CONFLUENCE OF AXIOMS

We now present the algorithm for testing the confluence of axioms for a set of axioms with certain restrictions. Theorems 4.4 and 4.5, which follow the presentation of the algorithm, demonstrate the validity of the method.

1. Ensure that there can exist no computation pathways of infinite length.

Construct a digraph from the terms of the L->R rewrite rules. Each node of the digraph represents either the left hand or right hand term of one of the axioms. Construct an arc from each of the left hand term nodes to its corresponding right hand term node. For each right hand term node, determine whether or not it provides a template match for any of the left hand term nodes, including its own. If there is a match, construct an arc from the right hand term node to the corresponding left hand term node. If the digraph has no cycles, the axioms are terminating. Otherwise there is at least one term that has an infinite computation path.

2. Ensure variables are preserved

If a variable appears on the right hand side of an axiom, it must appear on the left hand side and conversely. No axiom may be conditional.

3. Generate TESTSUITE(E)

Generate all the possible situations where two axiom templates may overlap:

Construct n expression trees corresponding to the n axioms of the specification.

```
For each expression tree:
  Traverse the tree in postfix
    If the node is an operator
      Find all trees whose root is equal to the operator
        node
          For each qualifying tree found
          Substitute the qualifying tree in place
            of the operator in the tree being tested
          If the original axiom still applies
            add the expression represented by the
              hybrid tree to TESTSUITE(E)
          Restore the tree
```

4. If TESTSUITE(E) is empty, we are finished. Otherwise, for each member of TESTSUITE(E), substitute some t in TERM(I) (of the appropriate sort) for each of the free variables.

5. Exhaustively compute each t in TESTSUITE(E). In other words, determine all of the computation pathways. If for each term, all of the computation pathways result in a common result then the axioms are confluent.

E. THEOREM 4.4

1. Statement

Let (S, E) be a specification.

E is a set of unconditional, terminating L->R rewrite rules such that variables are preserved.

If TESTSUITE(E) is empty, then the axioms are confluent.

2. Proof

Consider all of the complete computation pathways from t to some t'' in TERM(I). "Overlay" the n! computation pathways from t ->+ t' guaranteed by Theorem 4.3 on top of the complete computation pathways representing the exhaustive computation of term t. For those n! transformations, we know there are no "forks" to two or more terms which are in TERM(I) and not equal to each

other. Can there exist other pathways such that there are "forks" to two or more unequal, irreducible terms?

If the number of complete computation pathways from t to t'' in TERM(I) is equal to n!, then we know that the exhaustive computation pathways are exactly the pathways from t ->+ t' on the original APPLICABLE(t). Thus we know there were no "forks" resulting in more than one element of TERM(I). The number of exhaustive computation pathways cannot be less than n!, as those pathways are guaranteed by Theorem 4.3.

What if there are more than n! complete computation pathways? How can we be sure that the resultant terms for all of the pathways agree?

Note that Theorem 4.2 tells us that even if we do apply the new opportunites not part of the original set, the opportunities of the original APPLICABLE(t) must still apply. Theorem 4.3 tells us, for the n! pathways on the original terms, any new opportunities arising as a result of any of the original opportunities must still exist for the resultant term of those pathways.

Consider some intermediate term t',along some pathway from the original term t. If only members of APPLICABLE(t) have been applied so far, then we know that term t' is located along the set of pathways guaranteed by Theorem 4.3. Now, let us say that we have arrived at a term for which APPLICABLE(t') contains not only members of the original APPLICABLE set which have not been applied, but also some set of new axiom application opportunities. What do we know about this term? Theorem 4.3 guarantees that if we consider the entire APPLICABLE(t') set, for this intermediate term then there are | APPLICABLE(t') |! pathways corresponding to the set of axiom application opportunities (both new and old) that result in some common term t''. But this common term t'', must also be the same term to which you arrive had you followed the original n! pathways to their common term, t''', and then applied some permutation of the new opportunities that were part of its APPLICABLE(t''') set.

Therefore Theorem 4.3 not only guarantees that the n! pathways for the original term will result in a common term, but also guarantees that there cannot be a case where application of new axiom opportunities will cause a "fork" in the

exhaustive computation pathway to more than one term in TERM(I). This must be true for all terms in the specification. Thus the axioms are confluent, and TERM(I) is a canonical set. Figure 4.1 illustrates the construction for a term of an arbitrary specification for which TESTSUITE(E) is empty.

SPECIFICATION NONSENSE
is

SORT A

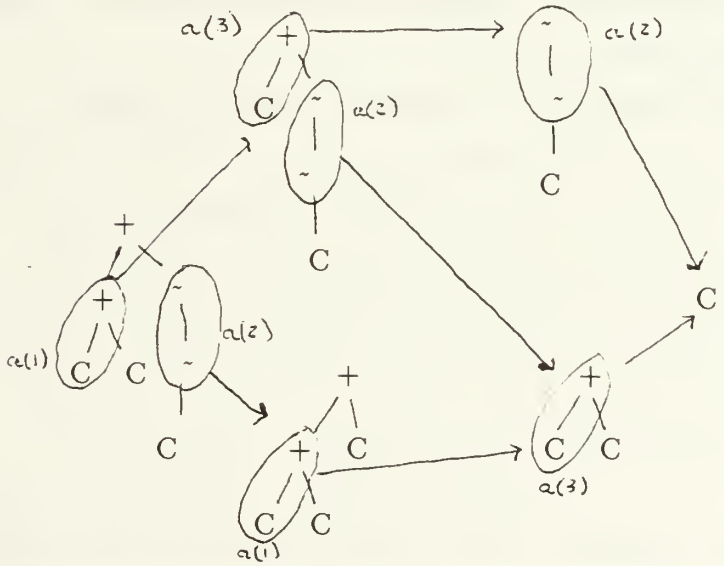
OPS

C: -> A
˜ : A -> A
+: A, A -> A

AXIOMS:

C + a = a
˜ (˜ (a)) = a

TESTSUITE(E) for this specification is empty
t = (C + C) + ˜ (˜ (C))



Confluent Pathways
Figure 4.1

F. THEOREM 4.5

1. Statement

Let (S, E) be a specification.

E is a set of unconditional, terminating L->R rewrite rules such that variables are preserved.

If TESTSUITE(E) is non-empty, and if for all t in TESTSUITE(E) where each free variable is unified with some t in TERM(I), every computation pathway for a given term produces the same t'' in TERM(S), then the axioms are confluent.

2. Proof

First, we should note that the terms in TESTSUITE(E) are themselves templates, and represent every possible overlap condition or interaction between 2 axioms that share template operators. Just as with the original templates of E, there is preservation of variables, and we know that nodes in the expression tree which are ancestors of the overlap template node will be unaffected during any computation path from this overlap template.

The difference is that now, if m axioms apply to a given expression tree, the application of one axiom may deny the opportunity to apply another. Theorem 4.3 does not apply.

The test of TESTSUITE(E) demonstrates that no matter how each of its terms are computed, the resulting terms for each term will agree. Thus for some arbitrary term, if there is a sharing of axiom templates in one of the subtrees, and we consider only that part of the expression tree, the resulting term (either in TERM(I) or TERM(R)) will agree for all pathways from the overlap subtree.

We know that to be true for all overlap cases. What about an arbitrary term which may have several overlap subtrees? In fact, there can be situtations where overlap templates overlap with each other. Can we show that no matter how a particular term is evaluated that it will be transformed to a single common term?

We submit the following construction. Consider the union of TEMPLATE(E), the actual set of templates for the L->R rewrite rules, with TESTSUITE(E). This set contains all of the situations where axioms may be

applied to subexpressions of some arbitrary term t. The expression tree can be partitioned into a set of distinct non-overlapping subtrees using the templates of this new set. Note that there may be more than one partitioning. For example, referring back to the spec for INTEGER, consider the following term:

P(S(P(S(0))))

There are several axioms templates in the union of TEMPLATE(INTEGER) and TESTSUITE(INTEGER) that apply to this tree:

P(S(x))
S(P(x))
P(S(P(x)))
S(P(S(x)))

Each of these cases have been examined and we know that for those templates, the resultant terms agree. Our arbitrary term may be partitioned into 2 occurrences of the axiom P(S(x)) = x, or 1 occurrence of P(S(P(x))) or 1 occurrence of S(P(S(x))). Intuitively, it should be obvious that since we have tested each overlap case as well as the original axioms, no matter how we compute this term, the resultant term will be the same.

We now have the means to construct a proof which is similar to the proof in the non-overlapping case. Each term may be partitioned into distinct, non-overlapping expression trees. Consider each template, whether an overlap or original template, to be 1 axiom application situation. Let n be the number of applicable axioms (either axiom or overlap). We can now use Theorem 4.3 to say that there exist n! pathways from this term to some common term where only the initially appearing axiom situations are addressed. From here on, the argument is the same as for the non-overlap situation. No matter how the term is computed, the intermediate terms may be partitioned in some way that transformation to a common term will be observed.

The fact that a given expression may be partitioned in more than way does not present any problems. Every overlap situation has been checked, so we know that no matter which one we choose at any given point along the computation pathway, there cannot be a point where some single term will "fork" to two different subterms.

G. EXAMPLES ILLUSTRATING THE CONFLUENCE ALGORITHM

Appendices A through D illustrate the application of the test for confluence to four arbitrary specifications. The test for confluence on the INTEGER specification illustrates the case where there was overlap of axiom templates and the algorithm successfully demonstrated its confluence. We look at the INTEGER specification because not only does it provide us with an example of the test, but also it is a familiar data type such that we know by other mathematical means that the axioms are correct. The other three examples are for arbitrary "NONSENSE" specifications. Although it may be argued that none of the NONSENSE specifications appear to model any useful system, we should note that the test for confluence makes no determination as to the "usefulness" or "meaning" of the system being specified, only that the axioms as stated by the author of the specification will result in a canonical set of terms. The question of whether or not the entity you describe is the entity you wanted to describe cannot be answered by such mechanical means as this axiom testing algorithm. That is an issue of axiomatics, and far beyond the scope of this discussion. The aim of the research was to find out if we could develop mechanical techniques to assist us in developing axioms such that we describe a computable system. The final pronouncement of whether or not some "meaning" has been modeled will be the task of techniques which will be presumably more sophisticated and undoubtedly less mechanical.

V. CONCLUSIONS

A. CURRENT POSITION

The research demonstrates that if the axioms of a formal algebraic specification are treated as left to right rewrite rules with certain restrictions, there is a practical algorithm to determine whether or not they are confluent. If confluence can be shown, then we know that our specification is practically computable. The firm result at present calls for the axioms to be restricted according to the following parameters:

- Variables must be preserved
- The axioms must be unconditional
- The axioms must be terminating

It is believed that the algorithm may be easily extended to include L->R rewrite rules for which the restriction concerning conditional axioms may be dropped. This will not be formally proved, but the reader may wish to consider the following. Our argument in demonstrating the validity of the algorithm depended on the fact that the variables of the axiom templates were preserved. Not only did this mean that the values bound to the variables during a transformation on an axiom opportunity were never affected, but it implied that the values of the variables could play no role in the determination of an axiom application opportunity.

Conditional axioms take into account all of the possible values for a free variable, and are applied only if the value of the free variable satisfies the condition of the axiom. For example, some specification might have the following axiom:

$$\tilde{}\,(x) + y = \tilde{}\,(x + y) \quad \{\, x \mathrel{!}= \tilde{}\ \}$$

It is not enough that there be a match between some subtree of an expression and the axiom template for a conditional axiom. There is an additional "test" that must be made to examine the value of the free variable x.

The algorithm for confluence tests only the interaction of axioms where there is template overlap. In a conditional axiom, one or more of the free variables are themselves templates in a sense, as their value does play a role in determining whether or not a particular axiom applies. It is believed that TESTSUITE(E) could be extended to the case where there are conditional axioms so that not only are operator overlap situations addressed, but also variable overlap situations for those axioms which are conditional.

It is proposed in the case where there are conditional axioms, that in addition to enumerating all of the ways that a single axiom may share operator template nodes with each of the other axioms, that for each conditional axiom template, each of the other templates is bound to the free variables of the axiom template. In other words, examine what happens to conditional axioms whose immediate descendents are other axiom opportunities of the specification. Thus we will be able to determine what happens in a situation where the transformation on an axiom opportunity might create the opportunity for a conditional axiom where one did not exist before.

Dropping the restriction that variables be preserved would open up the procedure to many common specifications which presently may not be tested by the algorithm. For example, the Boolean data type would not satisfy the restrictions, as there would be axioms such as

$$T \text{ or } x = T$$

Future work in this area should be directed toward examining whether or not this restriction may be dropped. Unfortunately, the present argument in demonstrating the validity of the procedure depends upon this restriction.

B. THE CHALLENGE

The algorithm to test for the confluence of an axiom system, albeit a very restricted one, is a heartening result if only for one reason. Formal specifications will only be of value if they may be used in practical situations. There are those who may argue that the power of the theoretical model is reduced by such a simplistic approach as that presented in this paper. That may well be true, and the author does not presume that such an approach could enjoy the same status

as the theory on which it is based. The algorithm is but a tool developed in an attempt to tap the resources promised by the foundational work in formal algebraic specifications. The material product of this research was a simple algorithm. However, the important result, at least as far as the author is concerned, is that we can develop tools that will help us to work in parallel with the theory. The challenge is to find increasingly more powerful tools that are closely aligned with the theoretical intent of formal algebraic specifications, yet that do not sacrifice practicality or utility in real applications.

APPENDIX A

Specification is INTEGER

Operators

0: -> A
P: A -> A
S: A -> A
+: A,A -> A

The Axioms for the INTEGER Specification:

1: P(S(x)) = x
2: S(P(x)) = x
3: +(P(x), y) = P(+(x,y))
4: +(S(x), y) = S(+(x,y))
5: +(0, x) = x
6: +(x, 0) = x

Test Suite for INTEGER Axioms

The free variables of the axioms are made distinct in the following examples only so that the progress of the algorithm may be followed, and the resulting TESTSUITE(E) terms understandable.

Axiom 1: aSP
Overlap:
Axiom 1: P(S(x)) = x
Axiom 2: S(P(x)) = x
bPSP
Axiom 2: bPS
Overlap:
Axiom 2: S(P(x)) = x
Axiom 1: P(S(x)) = x
aSPS
Axiom 3: cPd+
Overlap:
Axiom 3: +(P(x), y) = P(+(x,y))
Axiom 6: +(x, 0) = x
cP0+
Overlap:

Axiom 3: +(P(x), y) = P(+(x,y))
Axiom 1: P(S(x)) = x
aSPd+

Axiom 4: eSb+
Overlap:
Axiom 4: +(S(x), y) = S(+(x,y))
Axiom 6: +(x, 0) = x
eS0+

Overlap:
Axiom 4: +(S(x), y) = S(+(x,y))
Axiom 2: S(P(x)) = x
bPSb+

Axiom 5: 0f+
Overlap:
Axiom 5: +(0, x) = x
Axiom 6: +(x, 0) = x
00+

Axiom 6: g0+
Overlap:
Axiom 3: +(P(x), y) = P(+(x,y))
Axiom 6: +(x, 0) = x
cP0+

Overlap:
Axiom 4: +(S(x), y) = S(+(x,y))
Axiom 6: +(x, 0) = x
eS0+

Overlap:
Axiom 5: +(0, x) = x
Axiom 6: +(x, 0) = x
00+

TESTSUITE(E) for the INTEGER SPECIFICATION is

bPSP
aSPS
cP0+
aSPd+
eS0+
bPSb+
00+

Following is the output from the exhaustive computations of each of the terms, where irreducible term "0" is substituted for each of the free variables.

Original expression: 0PSP

Computed: 0P
Total number of paths for computation was 2

Original expression: 0SPS

Computed: 0S
Total number of paths for computation was 2

Original expression: 0P0+

Computed: 0P
Total number of paths for computation was 3

Original expression: 0SP0+

Computed: 0
Total number of paths for computation was 8

Original expression: 0S0+

Computed: 0S
Total number of paths for computation was 3

Original expression: 0PS0+

Computed: 0
Total number of paths for computation was 8

Original expression: 00+

Computed: 0
Total number of paths for computation was 2

The test for confluence for the INTEGER specification demonstrates that the axioms for the specification are confluent.

APPENDIX B

Specification is NONSENSE

Operators

C: -> A
/: A -> A
+: A,A -> A
*: A,A -> A

The axioms for the NONSENSE Specification are:

1: /(/(a)) + b = b + a
2: (a * b) + c = c * (a + b)
3: /(a * b) + c = a + (b * c)

Test Suite for Entered Axioms

Axiom 1: a//b+
Axiom 2: cd*e+

For this specification, TESTSUITE(E) is empty. The axioms are confluent

APPENDIX C

Specification is NONSENSE

Operators

C: -> A
/: A -> A
+: A,A -> A

The axioms for the NONSENSE Specification are:

1: C + a = a
2: /(a + C) = /(a)
3: /(a) + b = /(a + b)
4: /((a + C) + b) = /(a + b)
5: a + C = a

Test Suite for Entered Axioms

Axiom 1: Ca+
Overlap:
Axiom 1: C + a = a
Axiom 5: a + C = a
CC+
Axiom 2: bC+/
Original template shared
Axiom 2: /(a + C) = /(a)
Axiom 5: a + C = a
Overlap:
Axiom 2: /(a + C) = /(a)
Axiom 5: a + C = a
bC+/
Overlap:
Axiom 2: /(a + C) = /(a)
Axiom 4: /((a + C) + b) = /(a + b)
Axiom 5: a + C = a
Axiom 5: a + C = a
fC+C+/
Overlap:
Axiom 2: /(a + C) = /(a)
Axiom 1: C + a = a
Axiom 5: a + C = a
CC+/

Overlap:
Axiom 2: /(a + C) = /(a)
Axiom 3: /(a) + b = /(a + b)
Axiom 5: a + C = a
d/C+/
Overlap:
Axiom 2: /(a + C) = /(a)
Axiom 5: a + C = a
bC+/
Axiom 3: d/e+
Overlap:
Axiom 3: /(a) + b = /(a + b)
Axiom 5: a + C = a
d/C+
Overlap:
Axiom 3: /(a) + b = /(a + b)
Axiom 2: /(a + C) = /(a)
Axiom 5: a + C = a
bC+/e+
Overlap:
Axiom 3: /(a) + b = /(a + b)
Axiom 4: /((a + C) + b) = /(a + b)
Axiom 5: a + C = a
fC+g+/e+
Axiom 4: fC+g+/
Original template shared
Axiom 4: /((a + C) + b) = /(a + b)
Axiom 5: a + C = a
Overlap:
Axiom 2: /(a + C) = /(a)
Axiom 4: /((a + C) + b) = /(a + b)
Axiom 5: a + C = a
Axiom 5: a + C = a
fC+C+/
Overlap:
Axiom 4: /((a + C) + b) = /(a + b)
Axiom 5: a + C = a
fC+g+/
Overlap:
Axiom 4: /((a + C) + b) = /(a + b)
Axiom 5: a + C = a
fC+g+/
Overlap:
Axiom 4: /((a + C) + b) = /(a + b)
Axiom 5: a + C = a
fC+g+/

Overlap:

Axiom 2: /(a + C) = /(a)
Axiom 4: /((a + C) + b) = /(a + b)
Axiom 5: a + C = a
Axiom 5: a + C = a
fC+C+/

Overlap:

Axiom 4: /((a + C) + b) = /(a + b)
Axiom 1: C + a = a
Axiom 5: a + C = a
CC+g+/

Overlap:

Axiom 4: /((a + C) + b) = /(a + b)
Axiom 3: /(a) + b = /(a + b)
Axiom 5: a + C = a
d/C+g+/

Overlap:

Axiom 4: /((a + C) + b) = /(a + b)
Axiom 5: a + C = a
fC+g+/

Axiom 5: hC+

Overlap:

Axiom 1: C + a = a
Axiom 5: a + C = a
CC+

Overlap:

Axiom 3: /(a) + b = /(a + b)
Axiom 5: a + C = a
d/C+

The distinct terms of TESTSUITE(E) are:

CC+
bC+/
fC+C+/
d/C+/
d/C+
bC+/e+
fC+g+/e+
CC+g+/
d/C+g+/

Substitute for each of the free variables, the irreducible term /(C)

Original expression: CC+

Computed: C
Total number of paths for computation was 2

Original expression: C/C+/

Computed: C//
Total number of paths for computation was 5

Original expression: C/C+C+/

Computed: C//
Total number of paths for computation was 56

Original expression: C//C+/

Computed: C///
Total number of paths for computation was 7

Original expression: C//C+

Computed: C//
Total number of paths for computation was 6

Original expression: C/C+/C/+

Computed: C///
Total number of paths for computation was 19

Original expression: C/C+C/+/C/+

Computed: C////
Total number of paths for computation was 72

Original expression: CC+C/+/

Computed: C//
Total number of paths for computation was 3

Original expression: C//C+C/+/

Computed: C////
Total number of paths for computation was 21

We observe that for each member of the TESTSUITE set tested, there are no forks to irreducible terms. The axioms for this specification are confluent.

APPENDIX D

Specification is NONSENSE

Operators

C: -> A
/: A -> A
&: A -> A
+: A,A -> A

The axioms for the NONSENSE Specification are:

1: /(/(a)) = &(a)
2: C + a = a
3: /(a) + b = /(a + b)
4: a + /(b) = /(a + b)

Test Suite for NONSENSE Axioms

Axiom 1: a//
Overlap:
Axiom 1: /(/(a)) = &(a)
Axiom 1: /(/(a)) = &(a)
a///

Axiom 2: Cb+
Overlap:
Axiom 2: C + a = a
Axiom 4: a + /(b) = /(a + b)
Cg/+

Axiom 3: d/e+
Overlap:
Axiom 3: /(a) + b = /(a + b)
Axiom 4: a + /(b) = /(a + b)
d/g/+

Overlap:
Axiom 3: /(a) + b = /(a + b)
Axiom 1: /(/(a)) = &(a)
a//e+

Axiom 4: fg/+
Overlap:
Axiom 2: C + a = a
Axiom 4: a + /(b) = /(a + b)
Cg/+

Overlap:

Axiom 3: /(a) + b = /(a + b)
Axiom 4: a + /(b) = /(a + b)
d/g/+

Overlap:

Axiom 4: a + /(b) = /(a + b)
Axiom 1: /(/(a)) = &(a)
fa//+

For this specification, the elements of the TESTSUITE set are:

a///
Cg/+
d/g/+
a//e+
fa//+

For each of the free variables in the terms of TESTSUITE(E), substitute the irrdeducible term "C" and exhaustively compute each term.

Original expression: C///

Computed: C/&
Computed: C&/
Conflict -> C/& != C&/
Total number of paths for computation was 2

Original expression: CC/+

Computed: C/
Total number of paths for computation was 2

Original expression: C/C/+

Computed: C&
Total number of paths for computation was 5

```
-----------------------------------------

Original expression:  C//C+

Computed:  C&
Computed:  C&C+
                Conflict -> C& != C&C+
        Total number of paths for computation was 3


-----------------------------------------

Original expression:  CC//+

Computed:  C&
        Total number of paths for computation was 5
```

Thus we see that for this particular set of axioms, we find terms that reduce to more than one member of TERM(I). This set of axioms is not confluent.

LIST OF REFERENCES

Goguen, J. A., Thatcher, J. W., Wagner, E. G. and Wright, J. B., *An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types,* Current Trends in Programming Methodology IV, Data Structuring, R. T. Yeh, ed., Prentice-Hall, 1978.

Guttag, J. V. and Musser, D. R., "Abstract Data Types and Software Validation", *Comm. ACM,* Vol. 21, No. 12, Dec 1978, pp. 1048-1064.

Bergstra, A. and Tucker, J. V., "Initial and Final Algebra Semantics for Data Type Specifications: Two Characterization Theorems", *SIAM Journal of Computing,* Vol. 12, No. 2, May 1983.

Naval Postgraduate School, Tech. Report NPS52 84-022, *A Formal Method for Specifying Computer Resources in an Implementation Independent Manner,* Davis, D., Monterey, Ca., Dec 1984.

Yurchak, J. M., *The Formal Specification of an Abstract Machine: Design and Implementation,* Master's Thesis, Naval Postgraduate School Monterey, Ca., Dec 1984.

Lilly, N. L., *An Algebraic Specification Language and a Syntax Directed Editor,* Master's Thesis, Naval Postgraduate School, Monterey, Ca., Dec 1984.

Bundy, A., *The Computer Modelling of Mathematical Reasoning,* pp. 115-132, Academic Press, 1983.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Superintendent Attn: Library (Code 0142) Naval Postgraduate School Monterey, California 93943	2
3.	Chairman (Code 52) Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
4.	Computer Technology Programs (Code 37) Naval Postgraduate School Monterey, California 93943	1
5.	Daniel Davis (Code 52vv) Department of Computer Science Naval Postgraduate School Monterey, California 93943	5
6.	Rachel Griffin 12 Glendale Road Marblehead, Massachusetts 01945	5

Thesis
G7825 Griffin
c.1 An algorithm to test for confluence in a system of left to right rewrite rules.